Database Engineering Process Modelling

Didier ROLAND, Jean-Luc HAINAUT

Université de Namur, rue Grandgagnage 21
B-5000 Namur (Belgique)
phone: +32-81-72.49.85
fax: +32-81-72-49-67
{dro,jlh}@info.fundp.ac.be
http://www.info.fundp.ac.be/~dbm

Abstract

It can be shown that software process modelling, seen as a general software development control technique, can be applied to more specific components as well. In this paper, we propose a model, a language and a tool dedicated to database engineering.

As it is now natural, the model includes the concept of product (schemas and texts) and that of process. A process is intended to solve a class of problem (defined by the target requirements) and describes a class a transformations that apply on products to produce other products. Each process type is defined by its product flows (input/output) and by the strategy through which the enacter (the developper or the tool itself) can solve the problem.

The MDL language allows the method engineer to specify the product types and the process types of the method. A MDL text can be compiled and stored in the repository of the DB-MAIN CASE tool in order to customize this tool according to this specific method.

Résumé

La modélisation des processus d'ingénierie logicielle, vue comme une technique de contrôle de développement de logiciel, peut également être appliquée à des composants plus spécifiques. Dans cet article, nous proposons un modèle, un langage et un outil dédiés à l'ingénierie des bases de données.

Le modèle inclut naturellement les concepts de produits (schémas et textes) et de processus. Un processus est destiné à résoudre une classe de problèmes (définie par des besoins) et décrit une classe de transformations appliquées à des produits pour en déduire d'autres. Chaque type de processus est défini par son flux de produits (entrées/sorties) et par la stratégie à suivre par un opérateur (le développeur ou l'outil lui-meme) pour résoudre le problème.

Le langage MDL permet à un ingénieur de méthodes de spécifier les types de produits et les types de processus de la méthode. Un texte MDL peut être compilé et enregistré dans le référentiel de l'AGL DB-MAIN afin de personaliser cet outil en concordance avec cette méthode spécifique.

Keywords

Process modelling, database engineering, CASE tools

1. Introduction

Not surprisingly, process modelling can be considered a relevant concern in the database engineering realm as well. In particular, it can prove positive in two domains, namely methodological guidance and traceability. A CASE environment which has been made methodology-aware can guide practitioners in complex engineering processes, not only through a *what-to-do-next* check list, but also by explaining the rationale of each process and of each decision. Secondly, by keeping a formal track of all the documents, of all the processes and of all the decision, the CASE environment can maintain a precise documentation of the engineering process (the *history*) that can be consulted later on, and even processed in order to extract useful patterns (quality control, design patterns, heuristics, inverse history, etc). These aspects are particularly important in complex, non-standard processes in which exploratory practices are more frequent than in standard development activities.

1.1 The methodological assistance

During the last decades, methodologies such as Merise, Niam, OMT has become standard. Many organisations have also drawn up their own approaches, either fully specific, or, more frequently, as a customization of standard ones. These are expected to make the development coherent, complete, unambiguous and understandable by all the stakeholders. But, if they describe exactly what must be done, in terms of products for instance, they seldom specify how to do it. The goal of methodological assistance is to fill this hole.

The whole design of the database system will be called a process. It can be divided into sub-processes which correspond to some phases of the design. All these sub-processes may also be divided into sub-sub-processes and so on until a certain point where processes cannot be refined further because they correspond to primitive operations. Of course, a methodology cannot reduce to a pure automaton (otherwise it would have been implemented as a program). On the contrary, a methodology appears as recommendations, more of less strictly enforced, that instruct a human agent on which activities must be carried out in order to solve a class of engineering problems: normalization, refinement, optimization, translation, conceptualization, etc. Obviously, the developper want to keep much freedom in the exact way the problems are solved.

Each process is a product transformation. A product typically is a database schema, such as an SQL script, a COBOL source file, a report, that is any document that can be useful for an engineer to work on or any document that can be produced. A process could then transform a conceptual schema into a logical schema or a COBOL source file into a conceptual schema (reverse engineering).

In fact, the designer can see a methodology as a help card that tells him what to do, when to do it, and what tools are recommended.

1.2 Building a documentation

The second aspect of CASE-supported process modelling is the automatic production and maintenance of the trace of the activities and of the products related to an engineering project. This trace, or *history*, can be used as a documentation of the project. Furher maintenance, migration or extension of the software product will largely profit from such trace. For example, the rationale of a decision cannot be ignored when trying, later on, to modify the components resulting from that decision. Another characteristics of histories is their formal aspects, which make them processable. Indeed, (1) a history generally will need some polishing before being useable (trimming it from traces of trial&errors, discarded branches, loops, etc), (2) useful information can be extracted from a history (design quality, auditing, skill, design heuristics, ressource allocation, timing, etc), (3) new derived histories can be obtained, such as a fictive forward history built by inversing the history of a reverse engineering project [CAiSE-96], (4) propagating design changes through the design products can be automated, or at least assisted, by replaying the history of the former design.

1.3 Specific aspects of database engineering

Though database engineering shares many common concerns, models and techniques with software development processes, it exhibits some specific characteristics which will prove favourable as far as process modelling is concerned. Being narrower than system engineering in general, its study has lead to more refined methodological proposals, buth in the small (e.g. decomposition theory) and in the large (e.g. DB/IS design methodologies). Practically, this advance has materialized into the great variety[1] of CASE tools currently on the market.

In addition, non-standard processes such as optimization, reverse engineering, evolution, migration, maintenance and conversion, has been studied in greater detail, and with more concrete results than in the general system engineering domain.

1.4 CASE support

Process modelling would be an academic activity only if no CASE support were not developed. Few proposals exist so far, at least on the commercial market. The approach proposed in this paper is being implemented in the DB-MAIN CASE tool. This work is carried out in the DB-Process framework, a joint project of DB-MAIN ([Hainaut,94]).

1.5 Contribution of the paper

This paper presents a generic framework comprising the basic concepts that are necessary to specialize process modelling to the database engineering world. This framework is based on a balanced specification of both design products and engineering processes. A method description language (MDL) allows method engineers to describe in detail all the database engineering products analysts can use or generate as well as the way-of-working to perform each database engineering process. The paper also shows how a database engineering method specified in MDL can be enacted by a CASE tool (DB-MAIN).

---

[1] This variety is decreasing, but this is another story.

1.6 Organization of the paper

Section 2 will state the main concepts from which our approach will be built. Then, the concept of product model (Section 3), of product type (Section 4) and process type (Section 5) are developed. A process (instance) is defined as a history (Section 6). Finally, CASE support is briefly discussed in Section 7.

2. Concepts

The proposed design process modelling approach is based on a transformational approach according to which each *design process* transforms a (possibly empty) set of *products* into another set of products:

- a **product** is a document used, modified or produced during the design life cycle of the information system; as we focus specifically on database specification, we will describe mainly database **schemas** and database-related **texts**.
- a **design process** is described by the operations that have been carried out to transform the products; each operation is in turn a process; atomic processes are called *primitives*, while the others will be called *engineering processes*; each process is supposed to be goal-driven, i.e. it tries to make its output products compliant with specific design criteria, generally called *requirements* [MYLOPOULOS,92];
- reporting in a precise way (1) the operations carried out during a process, (2) the products involved, and (3) the rationale according to which they have been carried in that way, form the trace or the **history** of the process;

The history of a process must follow a predefined commonly agreed upon *way of working*, called a **method**. In other words, a history is an instance of a method. More precisely, a method is defined by *process types* and *product types*:

- a **product type** describes the properties of a class of products that play a definite role in the system life cycle; a product is an instance of a product type;
- a **process type** describes the general properties of a class of processes that have the same purpose, and that process products of the same type; a process is an instance of a process type;
- the **strategy** of a process type specifies how any process of this type must be, or can be, carried out in order to solve the problems it is intended to, and to make it produce output products that satisfy its requirements; in particular, a strategy mentions what processes, in what order, are to be carried out, and following what reasonings. Only *engineering process types* are defined by a strategy. *Primitive process types* are basic types of operations that are performed by the analyst, or by a CASE tool.

Several product types can be given the same, or similar, properties. Hence the concept of **product model**. A model defines a general class of products by stating the components they are allowed to include, the constraints that must be satisfied, and the names to be used to denote them. A product type is expressed into a product model[2]. These concepts are sketched in Fig. 1.

Fig.2 illustrates this architecture through three examples. In hierarchy **A**, *C++ programs* form the model of source texts obeying the C++ syntax. *Main* is the class of main programs, while *Invoice/v2.0* is a particular program from this class. Hierarchies **B** and **C** describe *data models*, *classes of schemas* and *particular schemas* according to the same three abstraction levels.

In the following sections, we will describe these three concepts in more detail, together with some aspects of MDL, a Method Description Language through which they can be declared.

3. Product model

An in-depth analysis of database engineering methodology exhibits both strong similarities and many specific aspects. What makes them similar, among others, is that, at each level of abstraction, they rely on some variant of popular specification models. However, instead of adopting such models as off-the-shelves components, most methods redefine and customize them according to the needs, culture and available technology of the business environment. In some sense, there are as many ERA, NIAM and OMT models as there are organizations that use them. Product models are to be considered as a way to precisely define what is exactly intended by each model used by the organization. In particular, it defines the concepts, the names to denote them and the rules to be used to build any product compliant with this model. Due to practical reasons, there are two kinds of products, namely schemas and texts.

---

[2] For practical reasons we did not find it necessary to define the concept of process model, at least in a first step, thus making the picture *inelegantly* asymmetrical. In particular, we identified a strong need for higher level abstraction above product types, while we found few convincing examples for process types.

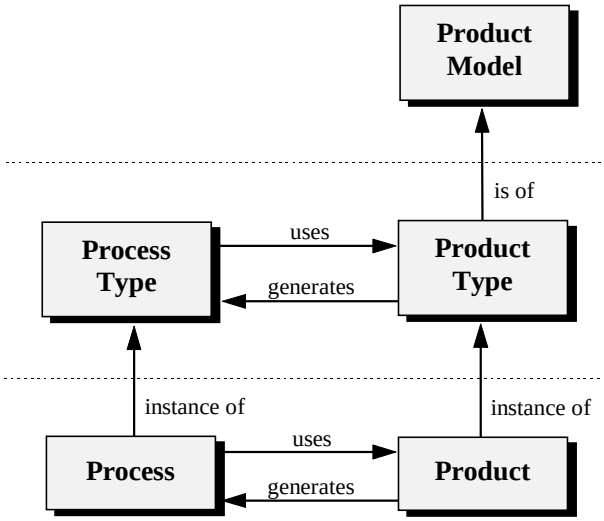

Fig. 1 *- The process modelling architecture*

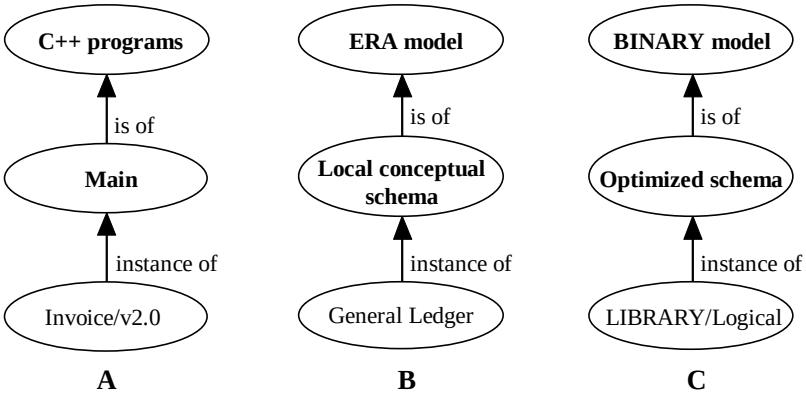

Fig. 2 *- Three examples of product hierarchies*

Abstraction levels	Representation paradigms
Conceptual	ERA, Merise Merise-OO, Chen, NIAM, OMT, Booch, Fusion, UML, etc
Logical	Relational, network, hierarchical, standard files, OO, etc
Physical	ORACLE v.7, SYBASE, IMS, IDS2, UDS, O2, GemStone, Microfocus COBOL, etc

Fig. 3 *- The 2-dimension scope of the GER model*

A **schema model** allows designers to specify data/information structures. A schema model is defined as a *specialization* of a generic ER model (GER). This wide-spectrum model is intended to describe data/information structures at different abstraction levels and according to the most popular paradigms (Fig. 3). It has been defined in [Hainaut,89].

The definition of schema model **M** comprises four main parts:

name: an identifier that will be used to reference **M** throughout the method description;

title: a more readable name of **M** that will be used by the supporting CASE tool user interface;

concepts: the list of the concept of the GER model **M** is made up of; their local name can also be specified. For instance, the GER concept of entity type will be included in an ER model defined in a French-speaking company (with name *Entité*), in an OO model (with name *Classe d'objets*) and in a relational model (with name *Table*). The concept of relationship type has no meaning in a relational model, and will not be included in the list of relevant concepts.

constraints: list of the rules that each schema expressed into **M** must satisfy. Each rule specifies a predicate that applies on a class of GER concepts, and that states what configurations are valid.

A constraint comprises the name of a predicate, the parameters and the diagnostic that have to be issued when a construct violates the predicate. The following table lists, in the MDL language [Roland,95], some examples of elementary predicates.

Elementary predicates (sample)

`ISA_per_SCHEMA (n m)`	for each schema, the number of is-a relations must be in the range [n-m]
`SUPER_TYPES_per_ISA (n m)`	for each entity type, the number of supertypes must be in the range [n-m]
`ATT_per_RT(n m)`	for each rel-type, the number of attributes must be in the range [n-m]
`ROLE_per_RT(n m)`	for each rel-type, the number of roles must be in the range [n-m]
`ALL_ATT_ID_per_ET (n m)`	for each entity type, the number of identifiers comprising attributes only is in the range [n-m]
`MAX_CARD_of_ATT (n m)`	for each attribute, the maximum cardinality must fall in the range [n-m]
`SUB_ATT_per_ATT (n m)`	for each attribute, the number of subattributes must be in the range [n-m]
`NONE_in_FILE_NAMES (f)`	no names in the schema can belong to the name list stored in file f

More complex predicates can be defined through boolean expressions, such as the following, which are part of the definition of a normalized ER model.

`COMP_per_EID (1 N)`
`  and ROLE_per_EID (0 0)`
`or COMP_per_EID (2 N)`
`  and ROLE_per_EID(1 N)`
	for each entity type identifier ID:
	either ID comprises 1 or several components and comprises no roles,
	or ID comprises 2 or more components and comprises roles.

`ROLE_per_RT (2 2)`
`or ROLE_per_RT(3 4)`
`  and ATT_per_RT (1 N)`
`or ROLE_per_RT(3 4)`
`  and ATT_per_RT (0 0)`
`  and ONE_ROLE_per_RT (0 0)`
	for each relationship type R:
	either R comprises 2 roles,
	or R is N-ary and has attributes
	or R is N-ary, has no attributes and has no *one* (i.e. [0-1] or [1-1]) roles

Fig. 4 illustrates the MDL definition of a simple binary model close to the historical Bachman model.

The specification of a **text model** can be simple when no syntax is enforced. Otherwise, the file including the BNF grammar of the contents of the texts is mentioned (Fig. 5).

4. Product type

A product type D is an identified document used and/or produced by engineering process type P. When an instance p of P is performed, it uses/generates an instance d of D. In some cases, p can involve several instances d_i of D, or several instances p_i of P can each involve an instance d_i of D. A product type is compliant with a product model, that defines which concepts, which names and which assembly rules can be used to make each instance of this product type (Fig. 6).

```
schema-model BACHMAN-MODEL
    title "Bachman binary model"
    description
        Simple Bachman model:   no supertype/subtypes structures,
                                binary one-to-many rel-types without attributes,
                                no compound attributes,
                                no multivalued attributes,
    end-description
    concepts
        project             "project"
        schema              "schema"
        entity_type         "record type"
        rel_type            "set type"
        role                "role"
        attribute           "field"
    constraints
        ISA_per_SCHEMA (0 0)        ; No is-a relations allowed
            diagnostic "Is-a relations are not allowed. Transform them"
        ROLE_per_RT (2 2)           ; Maximum degree of a rel-type = 2
            diagnostic "Rel-type &NAME must be binary. Transform it."
        ONE_ROLE_per_RT (1 1)       ; Only one "one" role (with card [i-1])
            diagnostic "Rel-type &NAME must have one ONE role. Transform it."
        ATT_per_RT (0 0)            ; Rel-types cannot have attributes
            diagnostic "Rel-type &NAME cannot havve attributes. Transform it."
        SUB_ATT_per_ATT (0 0)       ; Attributes must be atomic
            diagnostic "Attribute &NAME cannot have sub-attributes. Transform it."
        MAX_CARD_of_ATT (1 1)       ; Attributes must be single-valued
            diagnostic "Attribute &NAME must be single-valued. Transform it."
end-model
```

Fig. 4 *- MDL definition of a schema model*

```
text-model PLAIN-TEXT
    title "Plain ASCII text"
    description
      ASCII file that can be read by text editors
    end-description
    extensions "rpt", "txt"
end-model
```

```
text-model C++ programs
    title "C++ program"
    description
        C++ program according to the standard syntax
    end-description
    extensions "cpp"
    grammar cpp.bnf
end-model
```

Fig. 5 *- MDL definition of two text models*

5. Process type

A process type is the description of the activity that must/can be carried out to solve, in a general way, a class of problems. Besides general practical information (its name, its title, a short description, a help text), a process type is defined by its input and output product types, by its internal (local) product types and by its strategy. A strategy is the description of how a process instance can/must be carried out. It comprises mentioning the process types to perform and the way they can be carried out (the control flow). An important aspect of engineering process strategies is that they can range from completely deterministic to fully human-controlled. Consequently, the control structure offered by the process model must include both imperative and non-deterministic control structures. Through the analysis of a large collection of published, experimental and pragmatic methods, we have identified a small set of control structures that seem sufficient at the present time, but that still need evaluation. The proposal is fairly large and general though. In particular, it can describe in an elegant way unstructured toolbox-based approaches (*do all what you want, in the way you want, on any product*), completely deterministic procedures (*just choose the input product then click here*) and a large range of strongly- or loosely- constrained procedures.

The MDL specification of a process type **P** states the input/output flows of the process, as well as the way it must be carried out. It comprises seven parts:

```
product Optimized Schema
    title "Logical Optimized Schema"
    model BACHMAN-MODEL
    description
      Logical binary schema including
      optimization constructs
    end-description
end-product
```

```
product MAIN
    title "C++ Main program"
    model C++ programs
end-product
```

Fig. 6 *- Two product types*

name: an identifier that will be used to reference **P** throughout the method description;

title: a more readable name of **P** that will be used by the supporting CASE tool user interface;

explain: the section of a help file that explains the goal and the way of working of any process of type **P**;

input: list of the input product types of **P**;

output: list of the output product types of **P**;

update: list of the input product types that are updated by **P**;

internal: list of the product types that are created by **P** but have no existence outside **P**;

strategy: way of carrying out the instances of **P**;

A product type is given a local name that denotes its instances. It is associated with a product model. It can be declared with a *multiplicity* [i-j], stating the number of instances of that type that can be used during the execution of a process (j=N stands for "infinity"). For instance, in the declaration of a conceptual schema integration process, we can declare two input product types *master* and *secondary* both conforming with a conceptual model, the first one with multiplicity [1-1] represents the master schema and the second one with multiplicity [1-N] represents all the secondary schemas that will be integrated into the first one.

The strategy is declared in a semi-algorithmic way with the following control structures. In this description, *P* is a process type, *Si* is either a process type, as defined above, or one of the control structure defined hereafter and *C* is a condition as defined below.

do P: a process of type P must be performed;

toolbox T: use of the tololbox T as explained below;

S1;S2..;Sn: one instance of each process type must be performed in a sequential way;

each S1;S2..;Sn end-each: one instance of each process type must be performed, but in any order;

one S1;S2..;Sn end-one: one instance of one and only one process type must be performed;

some S1,S2..,Sn end-some: one instance instance of some process types can be performed;

if C then S1 else S2 end-if: if condition C is true, an instance of S1, otherwise an instance of S2, must be performed;

while C repeat S end-repeat: while condition C is satisfied, perform instances of S;

repeat S end-repeat until C: perform instances of S until condition C is met;

repeat S end-repeat: perform instances of S any number of times;

for V in T S end-for: perform S for each element (called V) of product collection T.

Some built-in process types are available, such as:

copy(T1,T2): create an instance of T1 with the same contents as T2 (from the same model);

cast(T1,T2): convert the instance of T1 into an instance of T2 (from different models).

Some structures use conditions. They can be of several types, among which:

- a strong condition is expressed as (the constraint part of) a product model; it states that its argument must comply with this model: `BINARY_MODEL(C)` means that the current instance of C **must comply** with the BINARY_MODEL;
- a weak condition is expressed as (the constraint part of) a product model as well; it states that its argument **should comply** with this model; it acts as a recommendation that can be ignored by the analyst: weak(`BINARY_MODEL(C)`) means that the analyst will be warned when the current instance of C does not comply with the BINARY_MODEL;

```
process COBOL_REVERSE_ENGINEERING
    title       "COBOL Reverse engineering"
    description Recovery of the refined logical schema and of the conceptual schema
                    of a set of COBOL files
    end-description
    explain     "COB_DBRE.HLP"
    input       Sources [1-N] : COBOL_PROGRAM
    output      LogicalSchema : COBOL_SCHEMA,
                ConceptSchema : CONCEPTUAL_SCHEMA
    intern      S : COBOL_PROGRAM,
                CobolSubSchemas [1-N] : COBOL_SCHEMA,
                P : COBOL_SCHEMA,
                GlobalCOBOL : COBOL_SCHEMA,
                PreConceptual : CONCEPTUAL_SCHEMA
    strategy
        for any S in Sources
            do COBOL_EXTRACTION(S,CobolSubSchemas)
        end-for;
        for any P in CobolSubSchemas
            do SCHEMA_ENRICHMENT(P,Sources)
        end-for;
        do INTEGRATE(CobolSubSchemas,GlobalCOBOL);
        copy (GlobalCOBOL,LogicalSchema);
        cast (GlobalCOBOL,PreConceptual);
        repeat
            one
                do DE-OPTIMIZATION(PreConceptual);
                do UNTRANSLATION(PreConceptual)
            end-one
        end-repeat  until ASK("Is this schema purely conceptual (Y/N)?","Y");
        repeat
            do CONCEPTUAL_NORMALIZATION(PreConceptual)
        end-repeat until weak(NORMALIZED_ER_SCHEMA(PreConceptual));
        copy (PreConceptual,ConceptSchema)
end-process
```

***Fig. 7** - A process type example*

- the last kind of condition is purely informal; it is made of a simple readable message to which the user has to give an answer: `ask("Is this schema normalized ? (Y/N)","Y")` displays the message, and evaluates to true when the analyst answers "Y".

When a process cannot be further decomposed into other processes, it can use a *toolbox*. A toolbox is a set of tools selected among the built-in functions and transformations of the supporting CASE tool. The strategy is simple: *use any number of tool instances, in any order*.

A process is defined into the MDL language, but can be given a graphical representation as well. The process types are represented with rectangles while product types are represented with ellipses. Arrows between process types show the control structures of the strategy while arrows between process types and product types represents the product flows

The example of Fig. 7 represents a simplified strategy for standard file reverse engineering [Hainaut,93]. Its graphical representation is in Fig. 8.

The user must choose some COBOL files in the Sources set and perform a data structure extraction on them, then enrich some of the generated schemas. All the resulting schemas are then integrated in a single COBOL schema. This last one is copied in LogicalSchema, an output product then type casted into a (pre-)conceptual schema to prepare for conceptualization. This last operation is performed by doing de-optimization and untranslation operations, one at a time, until the user decides the schema is a purely conceptual one. Finally, it is normalized and copied in ConceptSchema, the other output product.

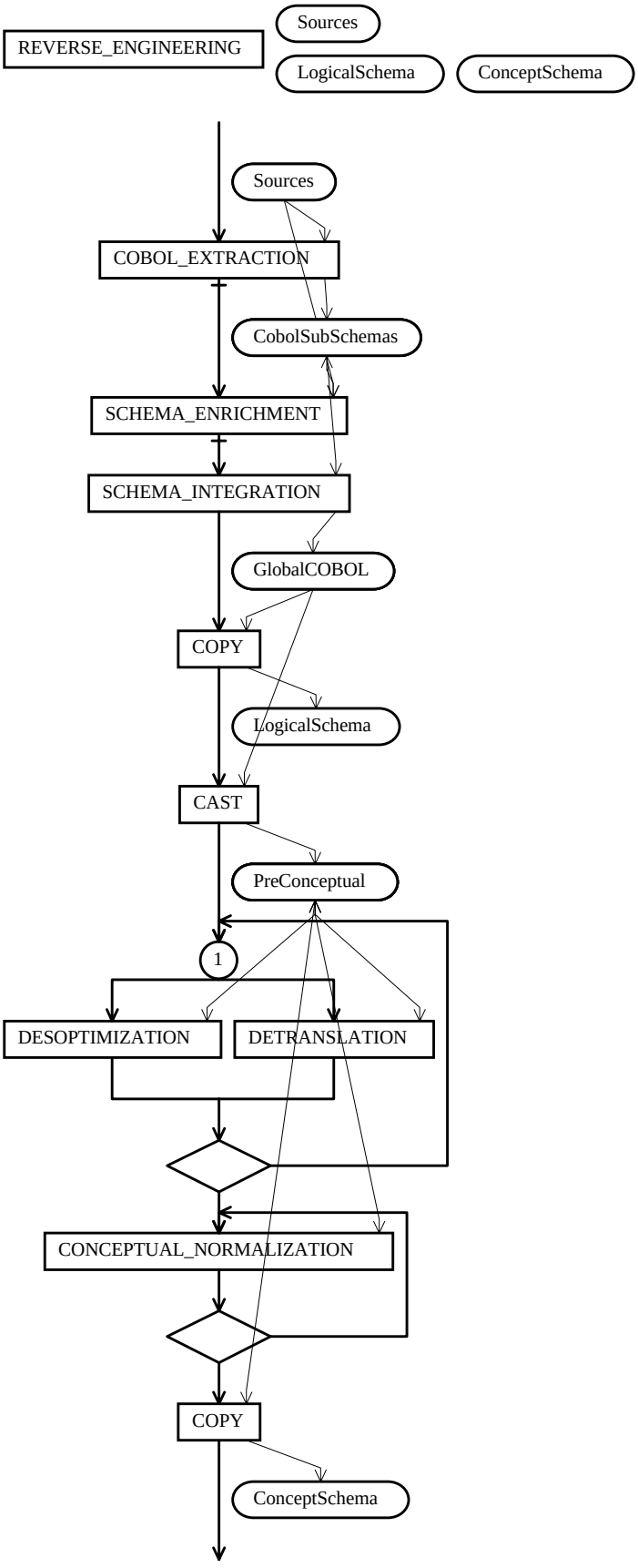

Fig. 8 *- The reverse engineering example graphical representation*

6. History

The history is the trace of an actual execution of an engineering process following its strategy. Technically speaking, it can be materialized by a sort of log file, a pure sequence of operations. But, a designer may be facing a choice between several ways of performing a process. He has to make hypotheses, each one reducing the problem to a particular context, and to solve the problem in each context. All the solutions are different versions of a product. The designer can take a decision a posteriori. All the hypotheses and all solutions must be recorded in the history as well as the decisions. Each hypothesis actually starts a new sequence of operations and each decision actually brings some branches to an end. Hence, the sequence of operations must be interpreted as a tree.

Now, let us consider the successful branches only. We remove all the branches corresponding to hypotheses which have not been retained, and whose end products have been discarded. Keeping the live branches only produces a **linear** history. This derived history is important since it describes the way the final products could have been obtained should the engineer have proceeded without any hesitation: replaying this history on the source products will yield the same output products as the actual process did.

The history is generated, maintained and presented by the supporting CASE tool. This should be done at different levels of aggregation, the extremes of which are:

- A **structured history**, which appears as an ordered tree in which each node represents a process instance. Leaves are primitive process instances, and non-leaf nodes are engineering process instances. The immediate children of node N represent instances of the processes mentioned in the script of the process of N. The root represents the instance of the main process, i.e. the project.
- A **flat history** shows the primitive process instances only. This concept is interesting because it is the easiest form of history to record. Indeed, since it represents no engineering processes, it is methodology-neutral, and can be built by simple CASE tools. In some situations, it could be the only form of history available. Such could be the case for loosely structured activities, such as some scenarios of reverse engineering.

7. CASE support

The process modelling approach described in this paper is being implemented as a methodological engine included in the DB-MAIN CASE tool [HAINAUT,94] and [HENRARD,96]. Its architecture is shown in fig. 9. The basic tools are the primitive functions of the CASE tool, from which the MDL toolboxes can be built. Additional customized tools can be built through the Voyager 2 meta-development environment. The assistants are advanced tools for helping the user to use the basic tools.

A method, described in MDL, can be compiled and loaded into the repository of the CASE tool. When a method is activated, the methodological engine controls the user interface in order to enforce this method. At any time,

- a view of the method shows what process types can be enacted with their specifications (input, output and updated product types) and description;
- allowed functions are enabled, the others are disabled;
- the schema model description scripts are transmitted to the schema analysis assistant to simplify schema validation;
- the history is recorded in the repository;
- the history can be visualized in a graphical way;
- tools are provided to browse through the history, to process it, and to replay it when necessary.

The architecture of the tool is depicted in Fig. 9. The lower layer consists in toolboxes providing basic services such as specification management and transformation. A collection of assistants proposes a knowledge-based assistance in complex and/or tedious processing, such as schema and text analysis, global transformation, schema integration and data reverse engineering. Most assistant provide for a scripting facility through which the analyst can develop libraries of reusable processes. The VOYAGER virtual machine is an interpreter on which complex Voyager-2 programs can be executed. This language is a 4GL that allows programmers to develop complex engineering functions such as analyzers, generators, specification transformation and analysis. From the process modelling perspective, basic services, the assistants and the Voyager-2 functions can be considered as a way to implement *basic deterministic processes*. The engineering processes, which are mainly controlled by analysts, are developed as MDL specifications. The MDL compiler introduces these specifications into the repository. From then, the methodological engine enacts the method by dynamically adjusting the user interface to the functionalities that are allowed, and by enforcing the specified control flow.

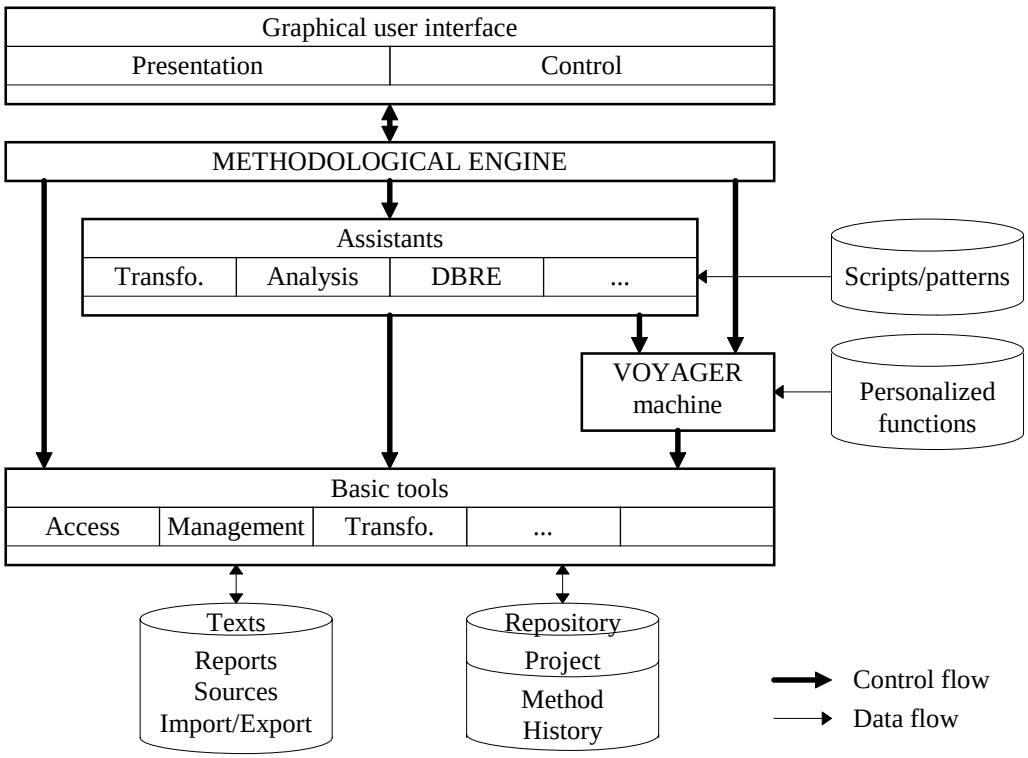

***Fig. 9** - The DB-MAIN architecture*

At the present time, the basic toolboxes, the VOYAGER virtual machine, seven assistants and the user interface are available. A prototype version that includes the MDL compiler, an extended repository that can accomodate method and histories, a graphical method inspector (see Fig. 8) and a log processor (prototype of the history manager) is evaluated. The methodological engine is under development.

8. Conclusion

Many research teams worked on different process modelling aspects along the years. At the end of the 80's, DAIDA, an ESPRIT project proposes a general model of the whole development of database centered information systems, including requirements modelling, the design and the implementation [JARKE,93]. This work proposed a expert system which is process oriented and explicitly dedicated to forward engineering. More recently (1992-1995), the Nature project centered its work on requirements engineering [NATURE,96]. In this project, requirement process modelling uses some concepts that are close to ours (products, processes, decisions, arguments, context,...), but they are used in a more declarative way. This work presents "a process engineering theory that promotes context and decision-based control of the development process." At the present time, the team of G. Grosz, C. Rolland, et al. [GROSZ,96] develops MENTOR, a computer aided requirements engineering environment that goes beyond Nature objectives by focussing mainly on the guidance engine. This work uses basic concepts that are similar to ours (similar terminology, similar architecture) but the specificity of requirements engineering leads this team to using a different approach with a non-algorithmic guidance engine that allow strategic or tactic guidance by use of an expert system. The team of J. Souquieres also works on requirements engineering process modelling [SOUQUIERES,93]. Their refinement development operators provide systematic techniques to produce software components. [CURTIS,92] and [GARG,96] both present different works coping with software development process modelling. They all have a common property: they are process oriented.

Despite its strong procedural aspects, which seem to aim it at defining imperative methods, our process model allows defining non-deterministic approaches. Indeed, many control structures are controlled guidelines for the analyst. In addition, *weak conditions* can help define flexible ways of working in which strategies are suggested but not strictly enforced.

The history of an engineering process is a formal baseline which can be used for many purposes:

- replaying the history for documentation purposes; it can be performed at several speeds from step by step to full speed between two logical points;

- undoing previous operations: by using the property that most transformations are reversible and applying their inverse; by recovering a previously saved state and replaying all operations performed since then; or by a combination of both these techniques;
- reverse engineering is the process of re-constructing a possible history that could have been used to produce the analysed products [HAINAUT,96b];
- database evolution is the process that consists in propagating a modification to one product to all other products in the history, upward and downward [HAINAUT,94].

When defining the scope of the approach, we concentrated on medium-term useability aspects. Quite naturally, this forced us to leave some important aspects aside, hopefully for further examination. Among others, two domains were intentionally ignored, namely goals and project management (actors, schedule, ressources), though they can be taken into account to some extent by the current architecture. Goals can be perceived at two levels, the *predefined goal* intentionally assigned to a process by the method engineer, and the *informal goal* the analyst has in mind when carrying out a process. Some kinds of predefined goals can be enforced provided they can be translated into product models. For instance, the goal "the output schema must be normalized" can be expressed by a product model allowing normalized structures only, and enforced by an adequate strategy the process type in charge of generating this product type. Informal goal can be implemented as informal annotations assigned by the analyst to process or product instances.

9. Bibliography

CURTIS,92 B. Curtis, M. I. Kelner, J. Over, *Process Modeling*, Communications of the ACM, September 1992, Vol.35 No.9.

GROSZ,96 G. Grosz, S. Si-Said, C. Rolland, *Mentor : un environnement pour l'ingénierie des méthodes et des besoins*, INFORSID'96, Bordeaux, 4-7 juin 1996.

HAINAUT,89 J-L. Hainaut, *A Generic Entity-Relationship Model*, in Proc. of the IFIP WG 8.1 Conf. on Information System Concepts : an in-depth analysis, North-Holland, 1989.

HAINAUT,94 J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland, *Evolution of database Applications : the DB-MAIN Approach*, in Proc. of the 13th Int. Conf. on ER Approach, Manchester, Springer-Verlag, LNCS 881, 1994

HAINAUT,95 J-L Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland, *Requirements for Information System Reverse Engineering Support*, in Proc. of the IEEE Working Conference on Reverse Engineering, Toronto, IEEE Computer Society Press, July 1995

HAINAUT,96a Hainaut J.-L., Englebert V., Henrard J., Hick J.-M., Roland D., *Database Reverse Engineering : from Requirement to CARE tools*, Journal of Automated Software Engineering, 3(2), 1996, Kluwer Academic Press.

HAINAUT,96b Hainaut J.-L., Henrard J., Hick J.-M., Roland D., Englebert V., *Database Design Recovery*, in Proc of the 8th Conf. on Advanced Information Systems Engineering (CAISE'96), Springer-Verlag, 1996.

HENRARD,95 Henrard, J., Englebert, V., Hick, J.-M., Roland, D., Hainaut, J.-L., *DB-MAIN: un atelier d'ingénierie de bases de données*, in Proc. of the "11èmes journées Base de Données Avancées", Nancy (France), September 1995.

HENRARD,96 Henrard J., Hick J.-M., Roland D., Englebert V., Hainaut J.-L., *Techniques d'analyse de programmes pour la rétro-ingénierie de base de données*, submitted to INFORSID'96, 1996.

JARKE,93 M. Jarke, editor. *Database Application Engineering with DAIDA*, Springer - Verlag, 1993.

MYLOPOULOS,92 J. Mylopoulos, L. Chung, B. Nixon, *Representing and Using Nonfunctional Requirements: A Process-Oriented Approach*, IEEE TSE, Vol. 18, No. 6, June 1992.

NATURE,96 Nature Team, *Defining Visions In Context: Models, Processes And Tools For Requirements Engineering*, Infirmation Systems, Vol. 21, No 6, 1996.

GARG,96 P. K. Garg, M. Jazayeri editors, *Process-Centered Software Engineering Environments*, IEEE Computer Society Press, Los Alamitos, California, 1996.

POTTS,88 C. Potts, G. Bruns, *Recording the Reasons for Design Decisions*, in ICSE 88, 1988.

ROLAND,95 D. Roland, *Un langage de description de processus : définition des prédicats structurels*, technical report, septembre 1995.

ROLLAND,93 C. Rolland, *Modeling the Requirements Engineering Process*, in 3rd European-Japanese Seminar on Information Modeling and Knowledge Bases, Budapest, May 1993.

ROSENTHAL,94 A. Rosenthal, D. Reiner, *Tools and Transformations - Rigourous and Otherwise - for Practical Database Design*, ACM TODS, Vol. 19, No. 2, June 1994

SOUQUIERES,93 J. Souquières, N. Lévy, *Description of Specification Developments*, in Proceedings of RE'93, San Diego (CA), 1993.

WANG,95 X. Wang, P. Loucopoulos, *The Development of Phedias: a CASE Shell*, Proceedings of the Seventh International Wokshop on Computer-Aided Software Engineering, Toronto, July 10-14, 1995.

YONESAKI,93 N. Yonesaki, M. Saeki, J. Ljungberg, T. Kinnula. *Software Process Modeling with the TAP Approach - Tasks-Agents-Products*, in 3rd European-Japanese Seminar on Information Modeling and Knowledge Bases, Budapest, May 1993.